

Option PF2, Automne 2016

« Programmation Scheme Avancée »

TP 1

Nous n'utiliserons plus le teachpack `valrose.rkt` ni le niveau de langage « Etudiant avancé », et travaillerons exclusivement dans des « modules ». Préparez un peu vos TP, ils ne durent que 1h30, et terminez-les en tout état de cause avant d'avoir la correction sur le Web, d'accord ?

Exercice 1.1 a) On s'intéresse aux expressions arithmétiques Scheme n-aires sur les quatre opérations `+ - * /`. Ces expressions ne contiendront que des constantes. Par exemple `(+ (* 2 3 4) 10 (- 6 (/ 8 2 2) 5))` dont la valeur est 33. Pour rester propre, nous programmons un petit type abstrait du pauvre sur ces expressions dont une grammaire est donnée à gauche :

<code><expr> ::= NOMBRE <noeud></code>	<code>(define noeud list)</code>
<code><noeud> ::= (<op> <expr> <expr> ...)</code>	<code>(define (feuille? expr) (not (pair? expr)))</code>
<code><op> ::= + - * /</code>	<code>(define operator car)</code>
	<code>(define arguments cdr)</code>

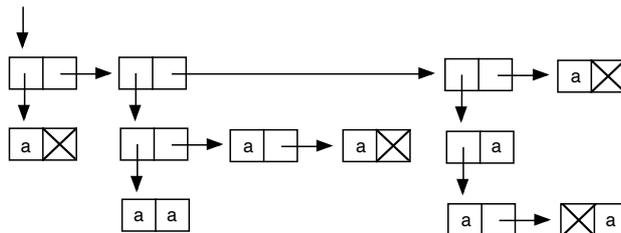
Un noeud est donc vu comme la donnée d'un opérateur à la racine et de la *liste des fils* (une liste des arguments de gauche à droite). Sauvegardez cet ADT de 4 lignes dans un module `adt-expr.rkt`. Attention, vous devez être dans le langage « déterminé par le source » débutant par `#lang racket`. N'oubliez pas le `provide`.

b) Fermez votre fichier `adt-expr.rkt`, et ouvrez un nouveau fichier `tp1.rkt`. Ce fichier sera aussi un module. Dans ce fichier, vous allez *requérir* le type abstrait `adt-expr`, et programmer une fonction (`valeur expr`) prenant une expression arithmétique et retournant sa valeur. Exemple : `(valeur '(+ (* 2 3 4) 10 (- 6 (/ 8 2 2) 5))) --> 33`. *Oui, je sais, je n'ai pas utilisé le constructeur de noeud, mais je ne le ferai pas dans une fonction manipulant ce type abstrait, promis juré, juste pour le test...*

• Vous restez dans le fichier `tp1.rkt` pour les exercices qui suivent.

Exercice 1.2 Avec les vraies fonctions Scheme sur les listes (`car`, `cdr`...), programmez une fonction (`beg L`) qui prend une liste `L` et dédouble tous ses éléments : `(beg '(il fait beau)) → (il il fait fait beau beau)`. Inversement programmez (`debeg LB`) telle que `(debeg (beg L)) ≡ L`.

Exercice 1.3 a) Linéarisez le diagramme ci-dessous de *boîtes et pointeurs* en une expression parenthésée. On prendra bien soin de livrer l'expression unique totalement simplifiée [sans point ou parenthèse superflus] :



b) Réciproquement, quel est l'unique diagramme de « boîtes et pointeurs » représentant l'implantation en mémoire de la liste : `((a b) . c) (d ((e . f))) g () h`

Au fait, à quoi voyez-vous que c'est une *liste* [dans l'écriture parenthésée ET dans le diagramme de boîtes et pointeurs] ? Il existe un endroit bien précis dans le dessin où cela se voit tout de suite...

Exercice 1.4 a) Ecrire une fonction `log-base` d'arité 1 ou 2. Avec un argument, (`log-base x`) retourne le logarithme népérien usuel de `x`, tandis que (`log-base x a`) retourne le logarithme $\log_a(x)$ de `x` en base `a`. Maths : $\log_a(x) = \ln(x)/\ln(a)$. Il vous faudra donc tester le nombre d'arguments.

`(log-base 10) → 2.3025...` `(log-base 10 2) → 3.3219...`

b) Plus difficile. Programmez une fonction (`minimax L1 L2 ...`) prenant un nombre quelconque de listes `L1, L2...` et retournant le plus petit maximum de toutes ces listes. Exemple :

`(minimax '(5 2 6 8 1) '(3 2 7 0) '(6 2 9 10 4)) → 7`

Exercice 1.5 Programmez la *macro* (`show expr`) prenant une expression SCHEME `expr`, et jouant le rôle de la fonction `show` de l'ancien teachpack `valrose.rkt`. Lors de l'exécution de `(show (+ 2 3))` dans l'éditeur, on souhaite voir au toplevel :

```
? (+ 2 3)
--> 5
```

• Vous verrez parfois dans les docs de Racket des **crochets** [et] à la place de certaines parenthèses, souvent pour dénoter des **couplets**. Ils ne sont pas obligatoires, on peut mettre des parenthèses à la place, c'est juste une histoire de style chez certains schemeurs. Vérifiez dans les docs de `let` ou de `cond` par exemple...

```
(let ([x 2] [y 3])
  (+ x y 10))
(cond [(> x 0) (* x 2)]
      [(< x 0) (/ x 2)]
      [else 8])
(for ([i (in-range 10 20)])
  (printf "~a " i))
```

Exercice 1.6 **IMPORTANT.** La **boucle for** est disponible en Racket, dans diverses formes s'inspirant plus ou moins de celle de Python¹. Elles est bien entendu programmée en Racket sous la forme d'une macro. Sans l'utiliser, programmez-en une version réduite nommée **For** avec un **F** majuscule. Vous allez uniquement simuler les deux formes possibles ci-dessous :

```
> (for ([i (in-range 10 20)])
  (printf "~a " i))
10 11 12 13 14 15 16 17 18 19
; for i in range(10,20) :
;   print(i,sep=' ',end='')

> (for ([i (in-range 10 20)] #:when (= 0 (modulo i 3)))
  (printf "~a " i))
12 15 18
; en Python ????

(define-syntax For
  (syntax-rules (in-range when)
    ((For ([x (in-range a b)]) e ...) (<une construction équivalente>))
    ((For ([x (in-range a b)] when test) e ...) (<une construction équivalente>))))
```

N.B. Les **compréhensions de listes** de Python sont des boucles `for/list` en Racket :

```
> (for/list ([i '(4 2 5 1 3)] #:when (odd? i))
  (sqr i))
(25 1 9)
```

Exercice 1.7 (Plus délicat) Voici une classe **Cons** en Python, dont les objets seront des doublets, à champs `car` et `cdr`.

```
class Cons :
    "La classe des doublets en Python"
    def __init__(self,car,cdr) :
        self.car = car
        self.cdr = cdr
```

Ajoutez à la classe `Cons` une méthode `__repr__(self)` chargé de retourner une chaîne de caractères contenant la représentation externe² du doublet `self` dans le style parenthésé de Scheme :

```
>>> doublet = Cons(1,Cons(Cons(2,Cons(3,4)),Cons(5,empty)))
# liste = chaînage de doublets
# dont le dernier cdr est vide
>>> doublet
(1 (2 3 . 4) 5)
```

POUR LES TP ULTERIEURS :

Il était bien pratique, ce teachpack `valrose.rkt`, au sens où nos petits utilitaires de programmation [et surtout les types abstraits] étaient automatiquement chargés dès l'entrée dans Scheme. Vous allez le remplacer par un module `utils.rkt` qui contiendra au fur et à mesure toutes les fonctions ou macros qui vous sembleront **ré-utilisables**. Aujourd'hui, vous allez le créer avec seulement la macro `show`.

Les deux premières lignes de votre prochain fichier `tp2.rkt` seront :

```
||| #lang racket
||| ;; fichier tp2.rkt
||| (require "utils.rkt") ; qui sera dans le même répertoire, c'est plus simple
```

Vous pourrez alors tester votre fonction `foo` préférée comme avant avec `(show (foo 5))` !... Mais n'oubliez pas de rajouter dans `utils.rkt` tout ce qui vous semblera intéressant au fur et à mesure des séances.

¹ C'est un peu la philosophie de Scheme. Si une construction linguistique d'un autre langage est intéressante, il est de bon ton de l'incorporer à Racket. Oui, mais en la programmant en Scheme (*Racket is a "programmable programming language"*)...